Quelques nouveautés C++ 11

CI-DESSOUS SONT INDIQUÉS LES LIENS SUR LE SITE OFFICIEL ET SUR LE SITE LTM

- Le **nullptr** : un nouveau mot clé pour mettre la valeur nulle dans son pointeur.
- **auto** : Les variables automatiques
- **atomic** : Les opérations atomiques (pas encore dans C++11)
- smart pointers : **unique_ptr** et **shared_ptr** site Stroustrup
- **&& (les types références rvalue)**
- **default & delete**
- **move and copy**
- **override** et **final**
- (**fonctions lamdba** site Stroustrup) (**fonctions lambda sur site ltm**)
- **threads** et **mutex**
- **std::async** (pas dans C++11 pour le moment)

thread

C++11 permet désormais la gestion des threads. Pour ce faire on utilise la classe *std::thread*.

Rapide rappel :
Les threads sont des unités d'exécution qui tournent dans un processus (PID). Chaque thread dans le système a un identifiant unique (id ou tid).

Chaque thread a une pile. Attention trop de threads dans un processus a tendance a fragmenter la mémoire.

Le fonctionnement des threads n'est pas déterministe. Exemple : Un thread1 dont on a demandé le départ avant un thread2 pourra démarrer après le thread2.

On peut fixer la taille de la pile du premier thread grâce à l'éditeur des liens dans un compilateur C ++.

Pour plus d'informations, je vous invite a vous reporter a de la littérature sur le sujet, facile à localiser sur internet.

Le C++11 propose un modèle classique proche de celui proposé par BOOST, ils sont désormais intégrés au langage. La mise en oeuvre est relativement simple.

1. La base

Exemple basique :

```
#include <iostream>
#include <thread>

void fct() {
  std::cout << "thread" << std::endl;
}

int main() {

   std::thread t1{fct};
   std::thread t2{fct};

   t1.join(); // attente de la fin de la tâche
   t2.join();

   return 0;
}
```

2. Passer des paramètres

Comment passer des paramètres à un **thread**, pratique courante en la matière ?
On utilisera la méthode std::bind(…), voir ci-dessous.

Voici :

```
#include <thread>
#include <vector>

void fct2(std::vector<int>& v) {
   std::cout << "thread fct2" << std::endl;
}

int main() {
  // 2. passage de paramètres
   std::vector<int> v(10);
   std::thread t3{ std::bind(fct2, v) };
   t3.join();

   return 0;
}
```

On remarquera ci-dessus que lors de la création de t3 on lui passe un objet v.

3. Synchroniser les tâches

Le fonctionnement des threads est non déterministe. On a souvent besoin de synchroniser les threads qui se partagent des ressources communes.

Le très classique **std::mutex** est ici utilisé (exclusion mutuelle). Le mutex propose des méthodes **lock()** et **unlock()** pour acquérir ou relâcher un mutex.

Du code, Du code ?

```
#include <iostream>
#include <thread>
#include <vector>

std::mutex m1;

void fct() {
   m1.lock();
   std::cout << "thread fct" << std::endl;
   m1.unlock();
}

int main() {

   // 1.
   std::thread t1{fct};
   std::thread t2{fct};

   t1.join();
   t2.join();

   return 0;
}
```

Ci-dessus la fonction **fct()** se voit pourvue d'un **Mutex** qui rend atomique l'exécution du **std::cout**, la console est cette fois synchronisée.

std::this_thread

Avec C++ et en incluant la bibliothèque <thread> on peut agir et récupérer des informations sur le thread courant.

std::this_thread::get_id() :
Retourne l'identifiant du thread

std::this_thread::sleep_for(std::chrono::milliseconds(2000)) :
Endort le thread courant 2 secondes.

Exercice sur std::thread

Créez un programme C++11 permettant la gestion d'une caisse de péage d'autoroute.

Les voitures sont représentées par des **std::thread**.
L'accès au péage doit être synchronisé.

Pour simplifier on imagine qu'il n'y a qu'une seule caisse pour payer.

Cas d'exemple : 3 voitures se présentent à la caisse du péage et payent chacune à leur tour.

Pour cet exercice vous utiliserez les classes **std:thread** et **std:mutex** ainsi que **std::this_thread** pour identifier et endormir le thread courant.